



# ML 10: STOCHASTIC GRADIENT, SYNTHETIC DATA, CEILING ANALYSIS

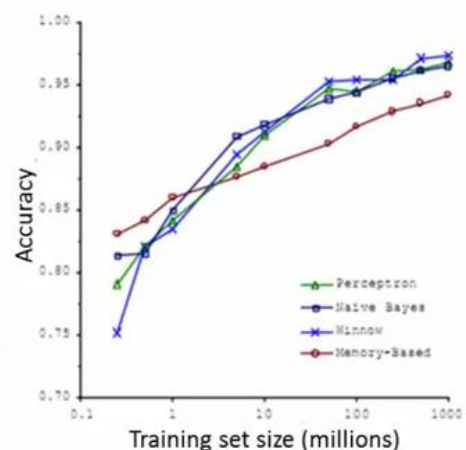
이번 주에는 *mini-batch*, *stochastic gradient descent*, *online learning*, *map-reduce* 등의 개념에 대해 배운다.

## Learning With Large Datasets

### Machine learning and data

Classify between confusable words.  
E.g., {to, two, too}, {then, than}.

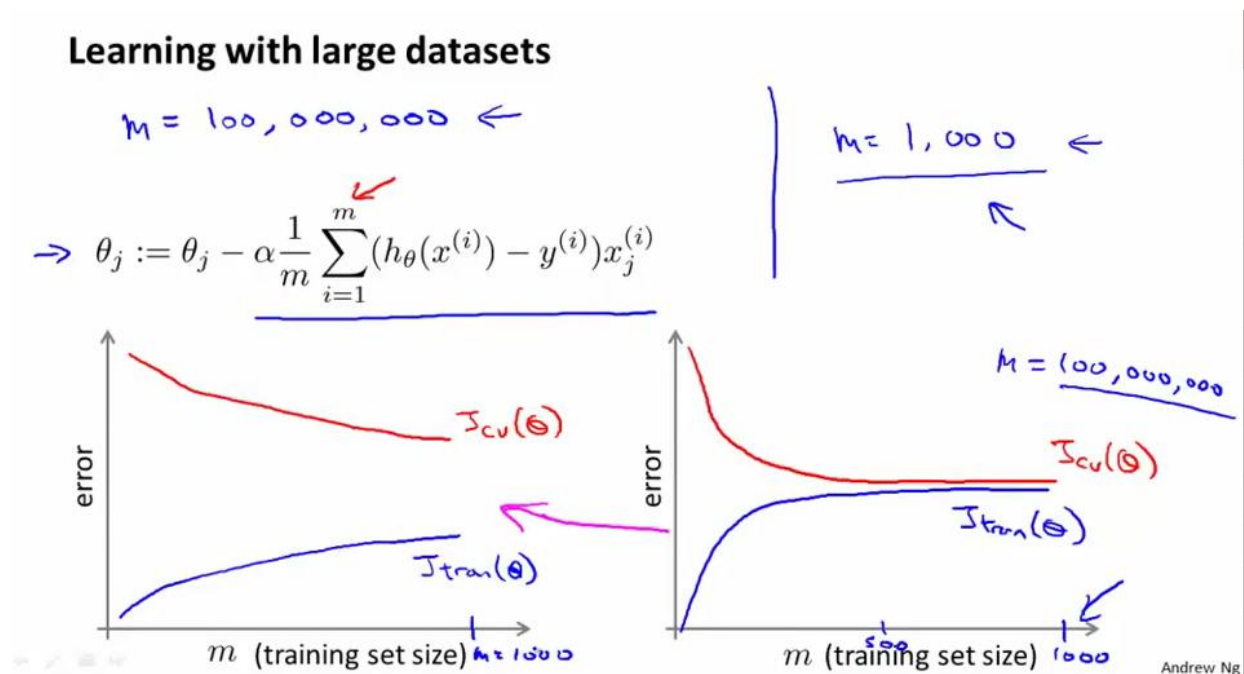
For breakfast I ate two eggs.



→ “It’s not who has the best algorithm that wins.  
It’s who has the most data.”

(<http://blog.csdn.net/linuxcunt>)

왜 그렇게 큰 데이터 셋이 필요할까? 좋은 퍼포먼스를 얻기 위한 한 가지 방법이, *low bias* 알고리즘에 *massive data* 를 활용해 훈련하는 것이기 때문이다.



(<http://blog.csdn.net/linuxcunt>)

그러나 커다란 데이터 셋을 연산하기 위해서는 연산비용이 정말 비싸다. 예를 들어  $m = 100,000,000$  이라 하면 *gradient* 를 계산하기 위해 매번  $k * m$  의 연산이 필요하다.

따라서 모든 데이터를 이용해 알고리즘을 훈련하기 보다는, 랜덤하게 고른 작은 서브셋에 대해서 알고리즘을 개발하고, 이후에 전체 데이터에 대해서 훈련하는 방법을 쓰기도 한다.

그러면  $m$  이 작아도 알고리즘이 충분히 잘 훈련된다는 것을 어떻게 보장할까? 이는 *learning curve* 를 그려보면 된다.

위 슬라이드에서 우측 하단은 *high bias* 알고리즘인데  $m$  을 많이 투입해도 별다른 이득이 없으므로 적당한 수준에서  $m$  을 정하면 된다.

물론 만든 알고리즘이 우측처럼 *high bias* 로 나오면, 좀 더 자연스러운 생각은 *hidden layer* 를 추가한다거나, 새로운 *feature* 를 도입해서 *bias* 를 낮추는 것이다.

# Stochastic Gradient Descent

## Linear regression with gradient descent

$$h_{\theta}(x) = \sum_{j=0}^n \theta_j x_j$$

$$J_{train}(\theta) = \frac{1}{2m} \sum_{i=1}^m (h_{\theta}(x^{(i)}) - y^{(i)})^2$$

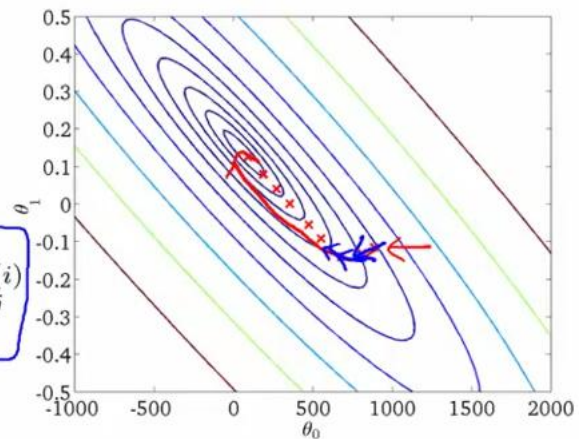
Repeat {

$$\theta_j := \theta_j - \alpha \frac{1}{m} \sum_{i=1}^m (h_{\theta}(x^{(i)}) - y^{(i)}) x_j^{(i)}$$

(for every  $j = 0, \dots, n$ )

}

$M = 300,000,000$   
Batch gradient descent



(<http://blog.csdn.net/linuxcumt>)

*gradient descent* 를 이용하는 *linear regression* 에서

$$h_{\theta}(x) = \sum_{j=0}^n \theta_j x_j$$

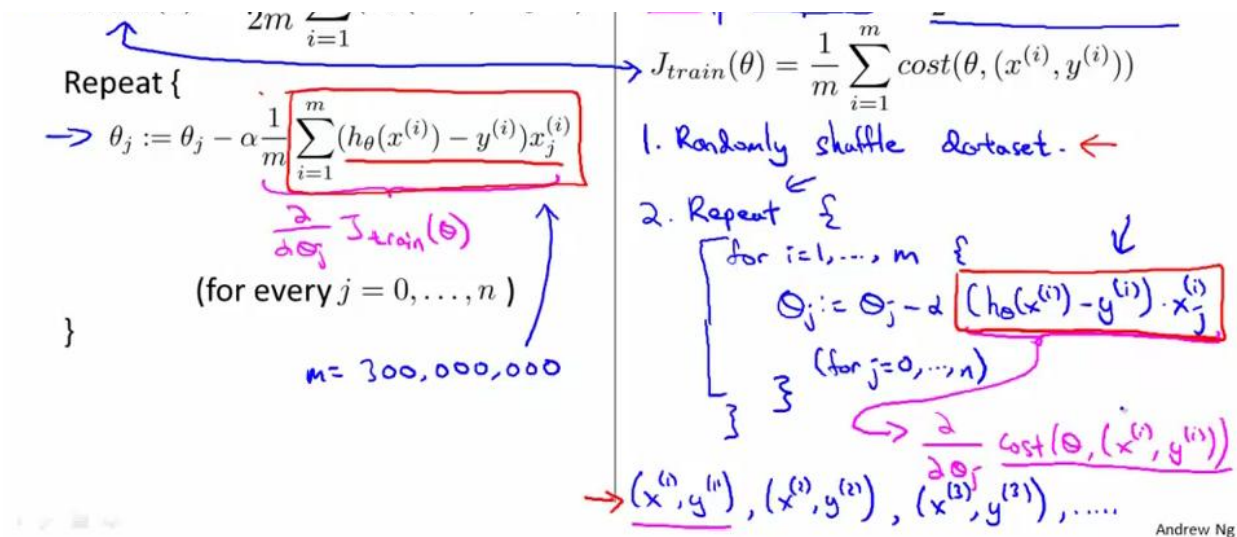
$$J_{train}(\theta) = \frac{1}{2m} \sum_{i=1}^m (h_{\theta}(x)^{(i)} - y^{(i)})^2$$

이미 언급했듯이 *batch gradient descent* 의 문제는,  $m$  이 크면  $J$  의 연산이 어마어마하게 많아진다는 것이다. 매 스텝마다  $m$  을 읽고, 계산값을 저장하기 때문이다.

### Batch gradient descent

$$\rightarrow J_{train}(\theta) = \frac{1}{2m} \sum_{i=1}^m (h_{\theta}(x^{(i)}) - y^{(i)})^2 \quad \left| \rightarrow \text{cost}(\theta, (x^{(i)}, y^{(i)})) = \frac{1}{2} (h_{\theta}(x^{(i)}) - y^{(i)})^2 \right.$$

### Stochastic gradient descent



(<http://blog.csdn.net/linuxcunt>)

이 문제를 해결하기 위해 *stochastic gradient descent*에서는 한 턴에 한 쌍의  $x, y$ 에 대해서만 *gradient*를 계산한다.

*batch gradient descent*에서는 한 턴마다 모든 모든 쌍의  $x, y$ 에 대해 *gradient* (=  $\theta_j$ )를 계산 했었다.

$$\text{cost}(\theta, (x^{(i)}, y^{(i)})) = \frac{1}{2} (h_{\theta}(x^{(i)}) - y^{(i)})^2$$

$$J_{\text{train}}(\theta) = \frac{1}{m} \sum_{i=1}^m \text{cost}(\theta, (x^{(i)}, y^{(i)}))$$

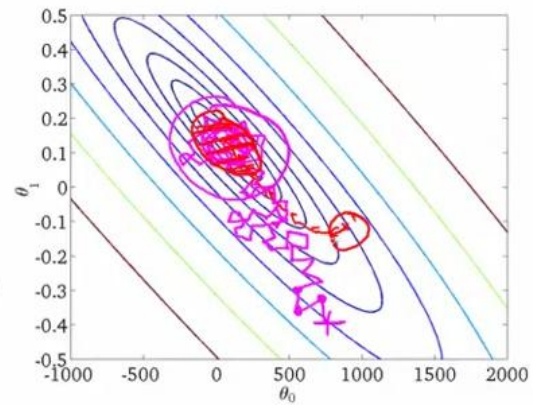
라고 하면

- Randomly shuffle dataset
- Repeat for  $i = 1$  to  $m$ 
  - $\theta_j := \theta_j - \alpha * \text{derivative of cost}(\theta, (x_i, y_i))$

즉  $J_{\text{train}}$ 의 미분 대신에  $\text{cost}$ 의 미분값을 이용해서 연산을 줄이는 방법이다. 이 때 데이터가 이미 랜덤하게 섞였다는 점을 기억하자. 기하학적으로 보면

## Stochastic gradient descent

- 1. Randomly shuffle (reorder) training examples
  - 2. Repeat { 1-10x
    - for  $i := 1, \dots, m$  {
      - $\theta_j := \theta_j - \alpha(h_{\theta}(x^{(i)}) - y^{(i)})x_j^{(i)}$   
(for every  $j = 0, \dots, n$ )
- $m = 300,000,000$



(<http://blog.csdn.net/linuxcunt>)

*batch*에서는 올바른 방향을 향해 가지만, 보폭이 좀 좁다. *stochastic*은 이리 갔다, 저리 갔다 하지만 결국에는 최저점을 향해 간다. 다만 *global optima*에 도달하지 못하고 그 근처에 도착할 수 있다.

m 이 굉장히 크면, *repeat* 부분의 루프를 1번만 돌려도 될 테지만, 작으면 여러번 돌려서 최대한 좋은 퍼포먼스를 내도록 훈련시킬 수 있다.

## Min-Batch Gradient Descent

### Mini-batch gradient descent

- Batch gradient descent: Use all  $m$  examples in each iteration
- Stochastic gradient descent: Use 1 example in each iteration

Mini-batch gradient descent: Use  $b$  examples in each iteration

$b = \text{mini-batch size. } b = 10.$

Get  $b = 10$  examples  $(x^{(i)}, y^{(i)}), \dots, (x^{(i+9)}, y^{(i+9)})$

$$\theta_j := \theta_j - \alpha \frac{1}{10} \sum_{k=i}^{i+9} (h_{\theta}(x^{(k)}) - y^{(k)}) \cdot x_j^{(k)}$$

$i := i + 10$





(<http://blog.csdn.net/linuxcunt>)

- **Batch gradient descent:** Use all  $m$  examples in each iteration
- **Stochastic gradient descent:** Use 1 example in each iteration
- **Batch gradient descent:** Use  $b$  examples in each iteration

$$\theta_j := \theta_j - \frac{\alpha}{b} \sum_{k=i}^{i+b-1} (h_{\theta}(x^{(k)}) - y^{(k)}) x_j^{(k)}$$

### Mini-batch gradient descent

Say  $b = 10$ ,  $m = 1000$ .

Repeat {

→ for  $i = 1, 11, 21, 31, \dots, 991$  {

$$\rightarrow \theta_j := \theta_j - \alpha \frac{1}{10} \sum_{k=i}^{i+9} (h_{\theta}(x^{(k)}) - y^{(k)}) x_j^{(k)}$$

(for every  $j = 0, \dots, n$ )

}

}

$m = 300,000,000$

↑

→  $b$  examples

→ 1 example

Vectorization

$b = 10$   
↑

(<http://blog.csdn.net/linuxcunt>)

$b$  는 보통 2 - 100 사이의 값이기 때문에 *batch* 보다 훨씬 빠르다. 또한 *vectorization* 을 이용하면 *gradient computation* 을 *partially parallelize* 할 수 있기 때문에 *stochastic* 보다 더 빠를 수 있다. 병렬화의 미덕

단점으로는 추가적인 파라미터  $b$  가 필요하다는 점이다. 그러나 *vectorization* 을 잘 이용하면 더 빨라지므로 문제 없다.

# Stochastic Gradient Descent Convergence

## Checking for convergence

→ Batch gradient descent:

→ Plot  $J_{train}(\theta)$  as a function of the number of iterations of gradient descent.

$$J_{train}(\theta) = \frac{1}{2m} \sum_{i=1}^m (h_{\theta}(x^{(i)}) - y^{(i)})^2 \quad M = 300,000,000$$

→ Stochastic gradient descent:

$$cost(\theta, (x^{(i)}, y^{(i)})) = \frac{1}{2} (h_{\theta}(x^{(i)}) - y^{(i)})^2$$

→ During learning, compute  $cost(\theta, (x^{(i)}, y^{(i)}))$  before updating  $\theta$  using  $(x^{(i)}, y^{(i)})$ .

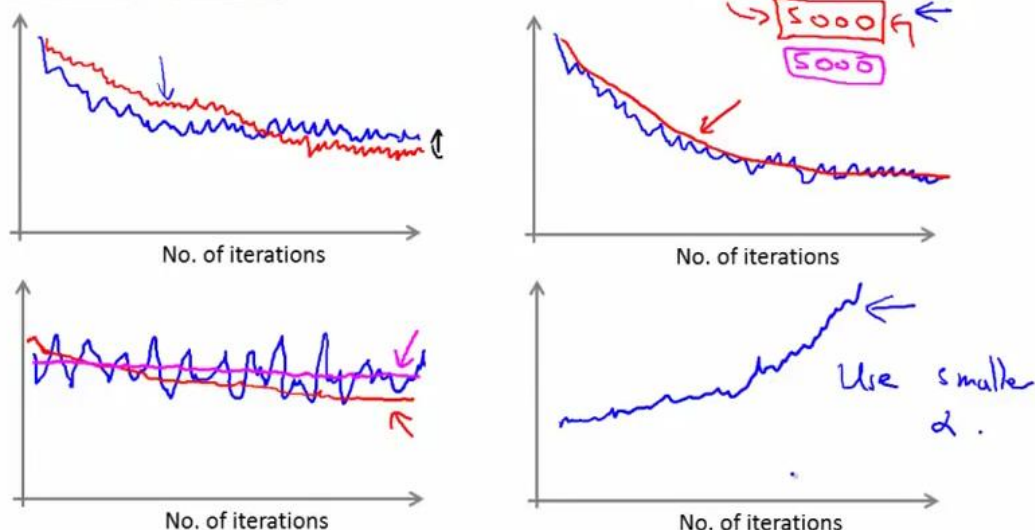
→ Every 1000 iterations (say), plot  $cost(\theta, (x^{(i)}, y^{(i)}))$  averaged over the last 1000 examples processed by algorithm.

(<http://blog.csdn.net/linuxcunt>)

*convergence* 를 검증하는 방법으로, 훈련시키는 동안 얻은  $cost(\theta, (x^{(i)}, y^{(i)}))$  평균 값을 이용해 플롯을 그릴 수 있다.

## Checking for convergence

Plot  $cost(\theta, (x^{(i)}, y^{(i)}))$ , averaged over the last 1000 (say) examples



(<http://blog.csdn.net/linuxcunt>)

*stochastic* 은 *global optimum* 을 찾아내지 못할 수도 있기 때문에, 그 주변에서 알짱거릴 수도 있다.

더 많은 `m` 을 투입하면, 까끌거리는 선보다 좀 매끄러운 곡선을 얻을 수도 있다.

때때로 알고리즘이 전혀 학습하지 못하는 것 처럼 보일수도 있는데, 그럴 경우 `m` 을 더 투입하면 좀 경사가 낮은 커브로 조금씩 *decreasing* 할 수 있다. 이를 보면 결국 훈련되긴 하는데, 평균값을 플랏으로 그리니 들쭉날쭉 해 보이는 것이다. (물론 학습하지 못하는 경우도 있다. `m` 을 더 늘려서 확인해 보자.)

`cost` 값이 증가한다면 더 작은 *learning rate* 값을 이용하자.

### Stochastic gradient descent

$$\text{cost}(\theta, (x^{(i)}, y^{(i)})) = \frac{1}{2}(h_{\theta}(x^{(i)}) - y^{(i)})^2$$

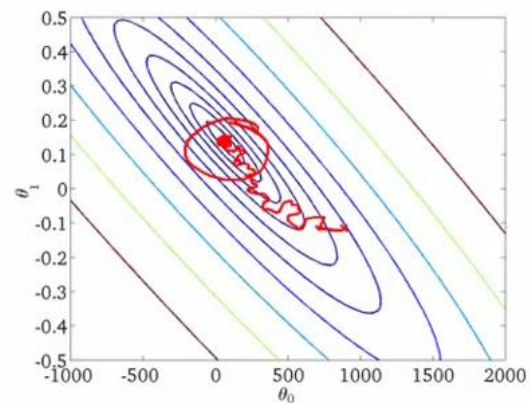
$$J_{\text{train}}(\theta) = \frac{1}{2m} \sum_{i=1}^m \text{cost}(\theta, (x^{(i)}, y^{(i)}))$$

1. Randomly shuffle dataset.
2. Repeat {
 

for  $i := 1, \dots, m$  {
 

$\theta_j := \theta_j - \alpha(h_{\theta}(x^{(i)}) - y^{(i)})x_j^{(i)}$ 

(for  $j = 0, \dots, n$ )



Learning rate  $\alpha$  is typically held constant. Can slowly decrease  $\alpha$  over time if we want  $\theta$  to converge. (E.g.  $\alpha = \frac{\text{const1}}{\text{iterationNumber} + \text{const2}}$ )  $\alpha \rightarrow 0$

(<http://blog.csdn.net/linuxcunt>)

*learning rate* 와 관련해서 위 슬라이드처럼 식을 만들면, 이터레이션 넘버가 천천히 증가하면서 `alpha` 가 감소해 *converge* 하는 결과를 얻을 수 있다.

If we reduce learning rate `alpha` (and run stochastic gradient descent long enough), it's possible that we may find a set of better parameters than with large `alpha`



# Online Learning

## Online learning

Shipping service website where user comes, specifies origin and destination, you offer to ship their package for some asking price, and users sometimes choose to use your shipping service ( $y = 1$ ), sometimes not ( $y = 0$ ).

Features  $x$  capture properties of user, of origin/destination and asking price. We want to learn  $p(y = 1|x; \theta)$  to optimize price.

Repeat forever {  
 Get  $(x, y)$  corresponding to user. price      logistic regression  
 Update  $\theta$  using  $(x, y)$ .  ~~$(x^{(i)}, y^{(i)})$~~   
 $\rightarrow \theta_j := \theta_j - \alpha (h_\theta(x) - y) \cdot x_j \quad (j = 0, \dots, n)$   
 }  
 $\rightarrow$  Can adapt to changing user preference.

online learning에서는 데이터를 얻어  $\theta$ 를 업데이트하는데 사용하고, 버린다. 큰 사이트라면 데이터가 지속적으로 들어오기 때문에, training data를 볼 필요가 없다. 다시 말해 같은 데이터를 두번 이상 쓰지 않는다는 말이다.

또 다른 장점으로 사용자의 취향 변화를 빠르게 반영할 수 있다는 점이다.

Can adopt to changing user preference

## Other online learning example:

### Product search (learning to search)

User searches for "Android phone 1080p camera" ←

Have 100 phones in store. Will return 10 results.

→  $x$  = features of phone, how many words in user query match name of phone, how many words in query match description of phone, etc.

→  $y = 1$  if user clicks on link.  $y = 0$  otherwise.

→ Learn  $p(y = 1|x; \theta)$ . ← predicted CTR

→ Use to show user the 10 phones they're most likely to click on.

$(x, y)$  ←  
 ↑ ↑

Other examples: Choosing special offers to show user; customized selection of news articles: product recommendation: ...

(<http://blog.csdn.net/linuxcunt>)

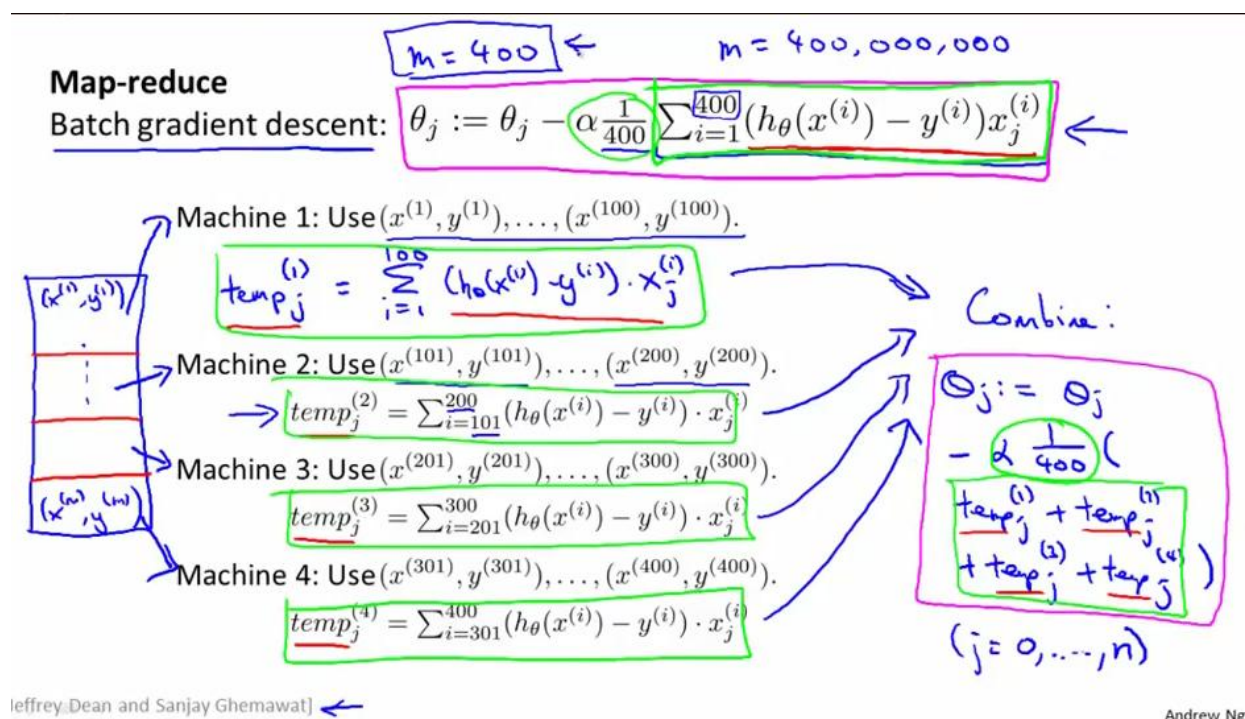
*product search* 에 *predicted CTR* 를 이용해, 검색어와 잘 매칭되는 상품을 결과로 돌려주 있다. 이때 매 검색마다 돌려주는 검색결과는 일종의 *training set* 이 된다.

- special offers
- customized selection

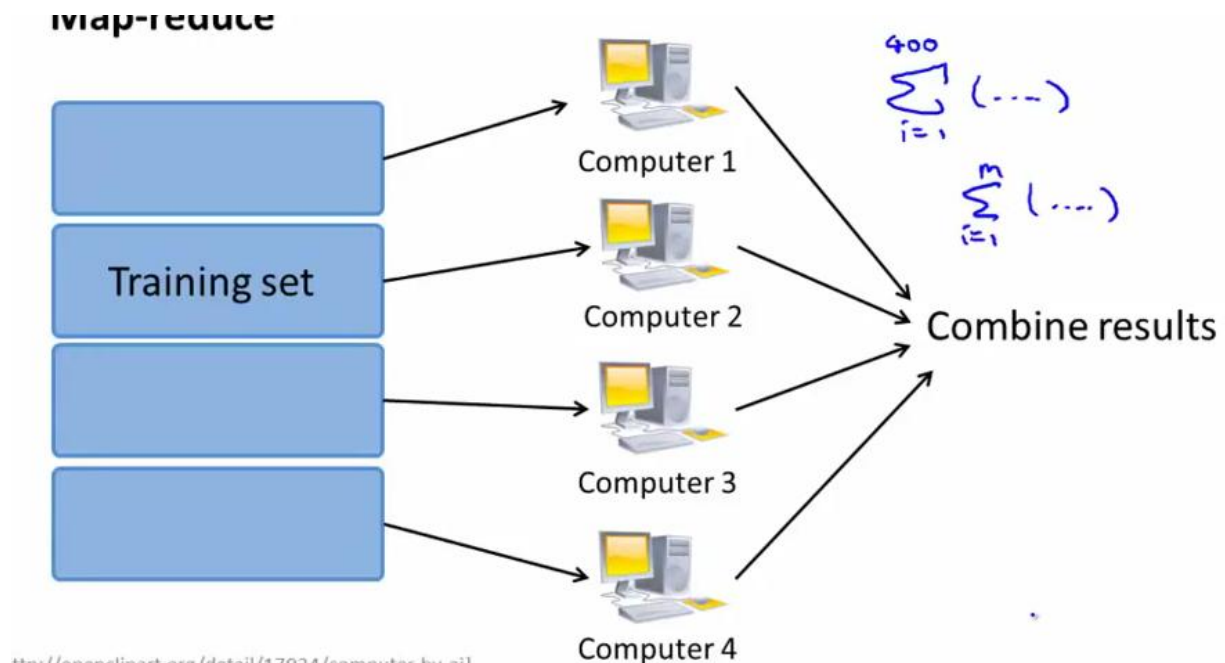
등에도 사용할 수 있다.

## Map Reduce and Data Parallelism

데이터가 어마어마하게 많으면, 하나의 컴퓨터에서 머신러닝 알고리즘을 돌리기가 좀 힘들다. 어떻게 해결할까?



Map reduce



쉽게 말하면, 분산해서 처리할 수 있는 결과는 **map** 으로 해결하고, 이 결과들을 이용해 전체적인 결과는 **reduce** 가 계산한다. (실제로는 **reduce** 도 여러개 일 수 있다)

### Map-reduce and summation over the training set

Many learning algorithms can be expressed as computing sums of functions over the training set.

E.g. for advanced optimization, with logistic regression, need:

$$\rightarrow \underline{J_{train}(\theta)} = -\frac{1}{m} \sum_{i=1}^m \underline{y^{(i)} \log h_{\theta}(x^{(i)}) - (1 - y^{(i)}) \log(1 - h_{\theta}(x^{(i)}))}$$

$$\rightarrow \underline{\frac{\partial}{\partial \theta_j} J_{train}(\theta)} = \frac{1}{m} \sum_{i=1}^m \underline{(h_{\theta}(x^{(i)}) - y^{(i)}) \cdot x_j^{(i)}}$$

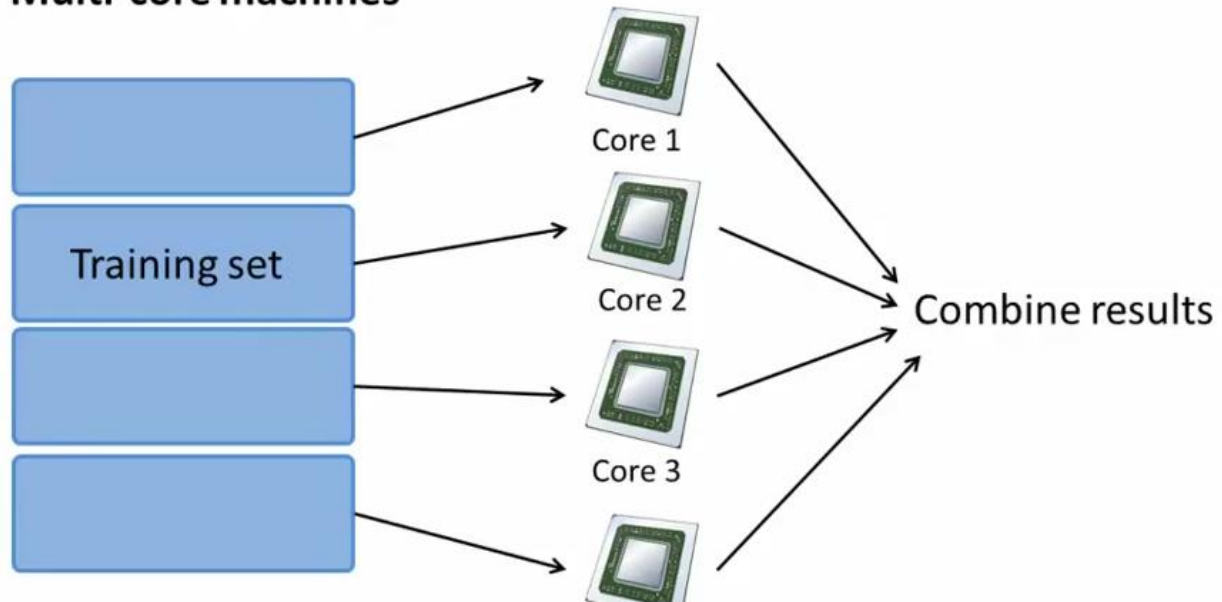
$\uparrow$   $\uparrow$   
 $temp^{(i)}$   $temp_j^{(i)}$

(<http://blog.csdn.net/linuxcunt>)

Many learning algorithms can be expressed as computing sums of functions over the training set

이렇기 때문에 *map-reduce* 가 큰 데이터셋에 대한 계산 처리 방법으로 좋은 해결책이 될 수 있다.

### Multi-core machines



(<http://blog.csdn.net/linuxcunt>)

요즘엔 대부분의 프로세서가 멀티코어이기 때문에, 하나의 컴퓨터에서도 병렬화를 이용해 계산을 빠르게 해 낼 수 있다. 이 경우는 *network latency* 등에 대해 생각을 안해도 된다. 참고로 좋은 라이브러리들은 자동으로 연산을 병렬화한다.

## Photo OCR and Pipeline

머신러닝 예제로 *Photo OCR* 을 알아보자.

### Photo OCR pipeline

→ 1. Text detection





→ 2. Character segmentation



→ 3. Character classification



~~Cleaning~~ → Cleaning

An

(<http://blog.csdn.net/linuxcumt>)

Photo OCR pipeline 은

- Image
- Text detection
- Character segmentation
- Character recognition

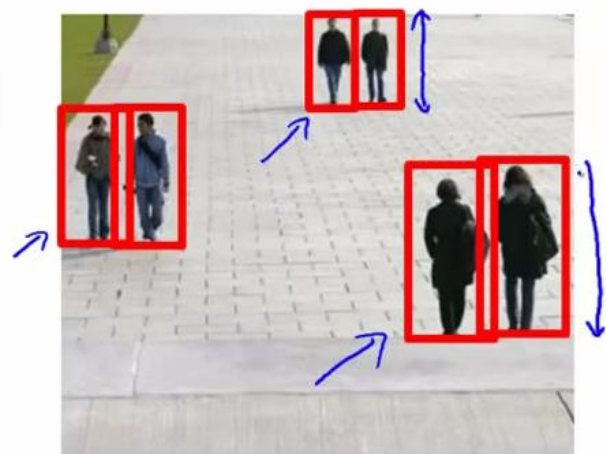
의 단계를 거친다. 각 단계마다 머신러닝을 적용할 수 있다.

## Sliding Windows

**Text detection**



**Pedestrian detection**



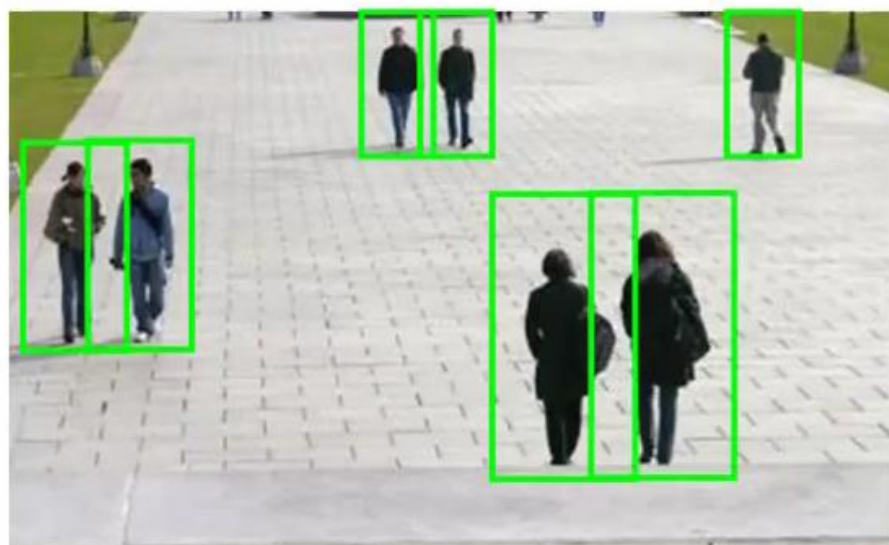
**Supervised learning for pedestrian detection**

1,000





## Sliding window detection



텍스트나, 보행자등 특정 패턴을 검색하기 위해 이동하는 *rectangle*의 단위를 *step-size, slide*라 부른다. *slide*의 사이즈를 변경해 가면서 패턴을 파악하는 방법을 *sliding window*라 부른다.

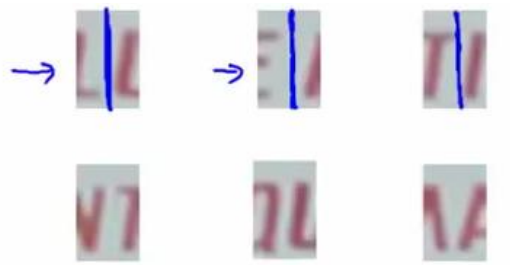
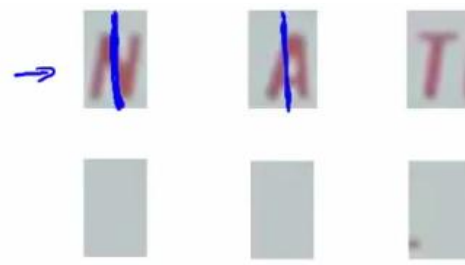
## Text detection

Positive examples ( $y = 1$ )Negative examples ( $y = 0$ )**Text detection**

(<http://blog.csdn.net/linuxcumt>)

텍스트를 인식해서, 근처의 텍스트와 묶는 *expansion* 작업을 하고 *character segmentation* 단계로 넘어간다.

**1D Sliding window for character segmentation**

Positive examples ( $y = 1$ )Negative examples ( $y = 0$ )

### Photo OCR pipeline

→ 1. Text detection



→ 2. Character segmentation



→ 3. Character classification

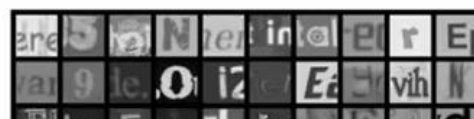


(<http://blog.csdn.net/linuxcunt>)

## Artificial Data

*low bias* 에 *massive data set* 을 조합하면 좋은 퍼포먼스가 나오긴 하는데, 어디서 커다란 데이터셋을 구할까? 작은 데이터 셋으로 커다란 데이터셋을 인위적으로 만들 수 있을까?

### Artificial data synthesis for photo OCR



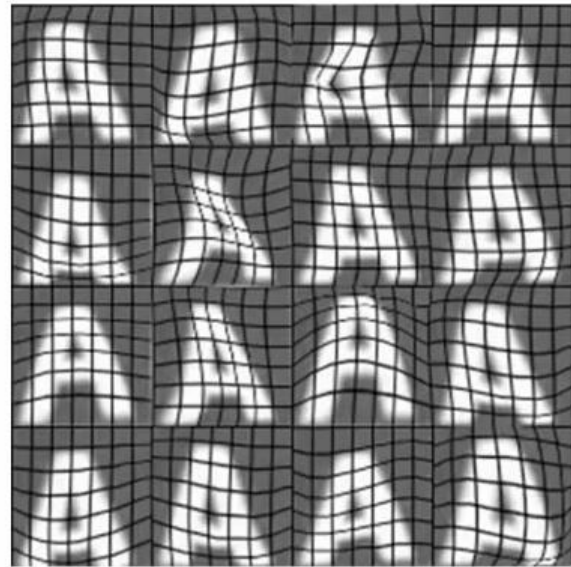
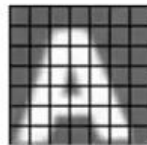


Real data



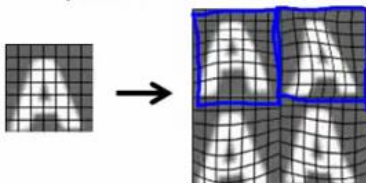
Synthetic data

## Synthesizing data by introducing distortions



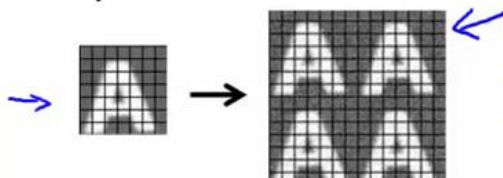
### Synthesizing data by introducing distortions

- Distortion introduced should be representation of the type of noise/distortions in the test set.



- Audio:  
Background noise,  
bad cellphone connection

- Usually does not help to add purely random/meaningless noise to your data.



- $x_i$  = intensity (brightness) of pixel  $i$
- $x_i \leftarrow x_i + \text{random noise}$



[Adam Coates and Tao Wang]

(<http://blog.csdn.net/linuxcunt>)

스케일링, 로테이션, 디스토션, 백그라운드 수정 등 다양한 조합을 통해 진짜처럼 보이는 *synthetic data* 를 얻을 수 있다. 마찬가지로, *speech recognition* 에도 *synthetic data* 를 만들어 퍼포먼스를 높일 수 있다.

## Discussion on getting more data

1. Make sure you have a low bias classifier before expending the effort. (Plot learning curves). E.g. keep increasing the number of features/number of hidden units in neural network until you have a low bias classifier.
2. “How much work would it be to get 10x as much data as we currently have?”
  - Artificial data synthesis
  - Collect/label it yourself
  - “Crowd source” (E.g. Amazon Mechanical Turk)

- *synthetic data* 를 만들기 전에 *low bias classifier* 인지 확인하자.
- 데이터를 조합하는데 들어가는 노력이 얼마나 들까 생각해보자
- *crowd source* 를 고려하자. (e.g. Amazon Mechanical Turk)

10 초당 1개의 *example* 을 수동으로 얻는다면, 10000 개를 얻는데 대략 3.5일의 시간이 걸린다.

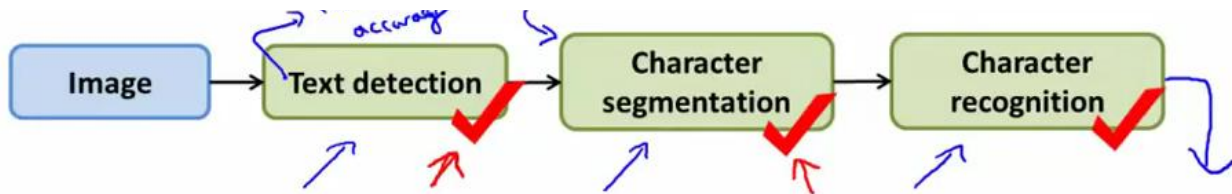
## Ceiling Analysis

이전의 *Photo OCR* 문제에서 퍼포먼스를 높이려면 파이프라인의 각 단계 중 어느 부분에 가장 많은 노력을 들여야할까?

### Estimating the errors due to each component (ceiling analysis)

100%





What part of the pipeline should you spend the most time trying to improve?

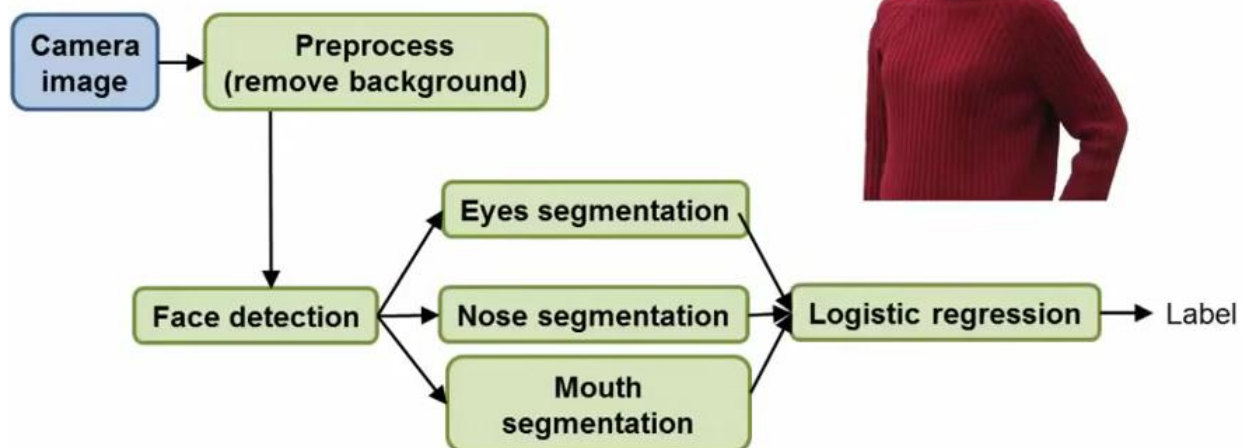
Component	Accuracy
Overall system	72%
Text detection	89%
Character segmentation	90%
Character recognition	100%

(<http://blog.csdn.net/linuxcunt>)

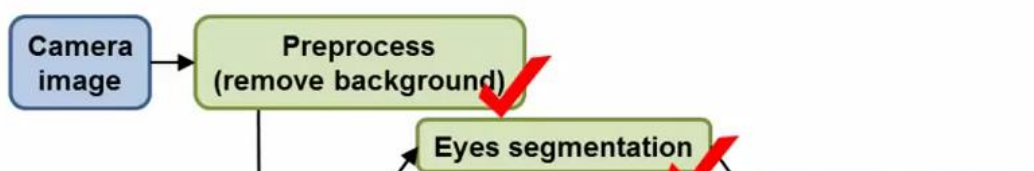
각 단계에서 수동으로 정확도 100% 를 만들었을때와, 전체적인 정확도를 비교해서 어느 부분을 향상 시켰을때 가장 효율적일지를 파악할수 있다.

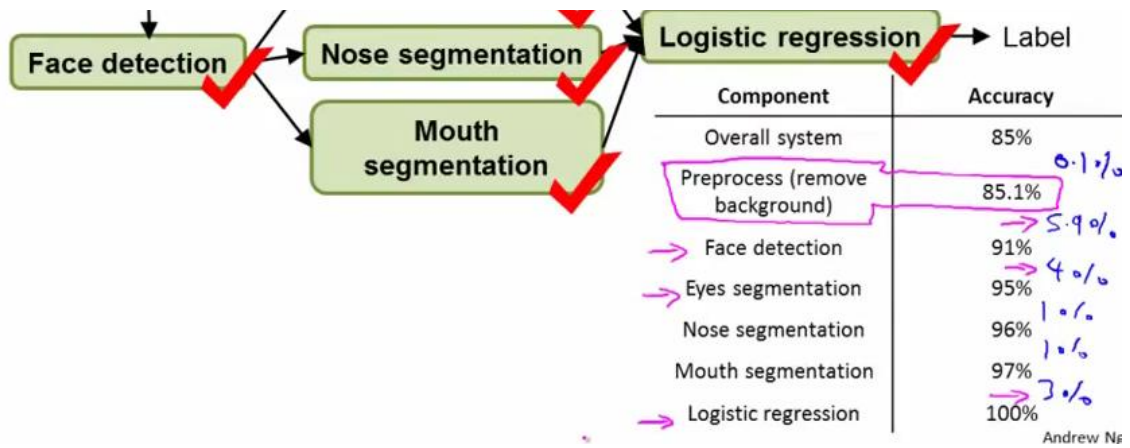
## Another ceiling analysis example

Face recognition from images  
(Artificial example)



## Another ceiling analysis example





(<http://blog.csdn.net/linuxcunt>)

## Summary

### Summary: Main topics

- Supervised Learning  $(x^{(i)}, y^{(i)})$ 
  - Linear regression, logistic regression, neural networks, SVMs
- Unsupervised Learning  $x^{(i)}$ 
  - K-means, PCA, Anomaly detection
- Special applications/special topics
  - Recommender systems, large scale machine learning.
- Advice on building a machine learning system
  - Bias/variance, regularization; deciding what to work on next: evaluation of learning algorithms, learning curves, error analysis, ceiling analysis.

(<http://blog.csdn.net/linuxcunt>)

supervised learning 의 종류로

- linear regresison

- logistic regression
- neural networks
- SVM

### unsupervised learning 으로

- k-means
- PCA
- anomaly detection

### 또한 머신 러닝의 응용으로

- recommender system
- large scale ML

### 마지막으로 머신러닝에 도움이 되는 주제로

- bias vs variance
- regularization
- evaluation technique
- learning curve
- error analysis
- ceiling analysis

등을 배웠다.



Thank you.

- Andrew Ng

## References

- (1) *Machine Learning* by **Andrew NG**
- (2) <http://blog.csdn.net/linuxcumt>
- (3) <http://blog.csdn.net/abcjennifer>

0 Comments    lambda

 Login ▾ Recommend 1     Share

Sort by Best ▾



Start the discussion...

LOG IN WITH

OR SIGN UP WITH DISQUS 

Be the first to comment.

ALSO ON LAMBDA

**Algorithm: Maximum Flow (Ford-Fulkerson)**

1 comment • 2 years ago

arkainoh — 오 프린스턴 세지윅 교수님의 Algorithm 강의로군요 저도 이거 예전에 들었는데 ㅎㅎㅎ 이부분이 좀 이해가 힘

**Substring Search Algorithm**

3 comments • 3 years ago

HyunSoo Park — 안녕하세요. KMP 알고리즘의 mismatch transition 부분에서 '무슨말이고 하니' 이 밑쪽부분 내용이 잘 이

**HOME**




1 comment • a year ago

Eunju Amy Sohn — 공부에 많은 도움을 받고 있습니다. 양질의 포스팅을 많이 올려주셔서 감사합니다!

**하스켈로 배우는 함수형 언어 4**

1 comment • a year ago

Junmo Roger Kang — 아이고 공부가 드디어 모나드에서 걸리네요 ㅜㅜ

 Subscribe     Add Disqus to your siteAdd DisqusAdd     Privacy

comments powered by Disqus

